Journal of Engineering and Applied Sciences 13 (Special Issue 2): 2858-2866, 2018
ISSN: 1816-949X
© Medwell Journals, 2018

# The Automated Weak Code Detection Tool for the Symbolic Execution-Based Vulnerability Analysis

[1]Sang-Kil Park, [1]Sung-Hwan Bae, [2]Jae-Pyo Park and [3]Yong-Joon Lee
[1]Department of IT Policy Management,
[2]Department of Information Security, Soongsil University, 07027 Seoul, Republic of Korea
[3]Department of Threat Response, Defense Security Institute, 03742 Seoul, Republic of Korea

**Abstract:** Software usage is increasing with the recent advancement of information technology which leads to an increased use of open-source software in various fields. However, as the use of open-source software that can be accessed by everyone increases, there might be potential problems regarding the vulnerabilities inherent in the open-source applications. In this study, we examined whether or not there are vulnerabilities in open-source software. To analyze the results, we suggested a technique of extracting the targets and their relevant areas of potentially weak source codes in terms of analyzing the vulnerabilities by means of symbolic execution. The suggested technique is as follows: a tree structure of the classes and methods within the source code of open-source software that is subject to security vulnerability analysis is created, then the DB of the target condition is established by identifying the data types and configuration patterns of the methods and it is examined through the system. If the condition is met, the corresponding method that is subject to vulnerability analysis is automatically extracted. The suggested technique involves the extraction and inspection of the weak source code which may pose high risk, instead of conducting a total inspection of source files. The suggested technique has an advantage in reducing the time of analysis and system load by means of the symbolic execution.

**Key words:** Vulnerability analysis, open-source, symbolic execution, auto-detection, weak-code, configuration

## INTRODUCTION

Most of the security problems that are constantly occurring and causing social issues are caused by security vulnerabilities inherent in the software which mostly result from mistakes made by the developers when coding the program in the process of software development (Khalili *et al*., 2016, Zhejun *et al*., 2013).

The damages caused by cyber-attacks are constantly increasing as use of personal computers and the internet increases. Defense systems are established with various security solutions that are introduced to protect computers from cyber-attacks and minimize damage but the scales and frequencies of security issues are continuously increasing. About 75% of the constantly increasing cyber-attacks result from the inherent weakness of software (Allodi and Massacci, 2014; Ernst, 2003). A typical example is the heart bleed bug of open-source OpenSSL which is used by about two-thirds of the world's websites. Also, MS (Microsoft) has reported that developers may reduce a significant amount of development cost if they consider the security factors beginning from the initial developmental stage, instead of making security patches (Microsoft Corporation, 2016).

For these reasons, finding the inherent vulnerabilities in the software development stage and taking measures is a good way to reduce cyber-attacks as well as to significantly reduce the development cost. It is also important to make continuous efforts to detect vulnerabilities of the source codes and applications not only in the development stage but also after they are already developed and in operation.

The techniques to detect vulnerabilities involve the static analysis method which is based on source code and the dynamic analysis method which is based on the execution system. However, false detection, undetection and over-detection are still occurring because of the limitations of analytical accuracy.

In this study, we tried to address the disadvantages such as over-detection and long analysis time of the conventional methods of open-source vulnerability analysis. To do this, we suggested a technique of designating the targets and the area of the source codes

**Corresponding Author:** Jae-Pyo Park,Department of Information Security, Soongsil University, 07027 Seoul, Republic of Korea

having potential risks while making an analysis by symbolic execution. A tree structure of the classes and methods within the source-code of open source software that is subject to security vulnerability analysis is created. The DB of the target condition is then established by identifying the data types and configuration patterns of the methods and is examined by means of the system. If the condition is met, the corresponding method that is subject to vulnerability analysis is automatically extracted. A class defines attributes and behaviors belonging to the same kind of group in object-oriented programming. A method is a concept corresponding to a function. It is defined in a class and defines the operations that can be performed on the object.

The proposed technique is to extract and examine the methods with high risk, instead of making a total inspection of source files when analyzing the security vulnerabilities by means of symbolic execution of open-source software. This technique has the advantage of reducing over detection and analysis time because it extract and inspects only the weak codes by analyzing the byte-codes of the source file, hierarchy tree generation, extraction of initial analytical targets, pattern matching-based target method extraction and analysis priority method list generation in order to extract the methods having high risk.

## Literature review

**Vulnerability:** Software with vulnerability problems has security issues that could lead to an unwarranted operation, access to specific data by ignoring the restrictions or information service denial attacks by unauthorized users which may result in information leakage or system destruction. Vulnerability analysis refers to the method of examining and analyzing software vulnerabilities to prevent accidents caused by security issues (Ransbotham and Mitra, 2013, Zerkane *et al*., 2016).

**Symbolic execution:** Symbolic execution is a technique for checking whether there is a problem by assigning symbol values to input variables in a software program unit instead of character values which allows program analysis by transferring inserted symbolic values to the operation numbers within the formula. The resulting symbolic formula has advantages in that it is simplified to express all the intermediate calculations and judgments in terms of symbolic insertion (Chen *et al*., 2013, Kebbal, 2006) and in that when the program meets a branch statement, it follows both paths to generate symbolic values and checks the conditions of various statements to find out at which point of the program each variable has certain values, so, it is possible to find vulnerability locations of the codes without actually operating the software source codes.

**Analysis based on symbolic execution:** Symbolic execution's basic principle is that a symbol is used to represent a specific value. Symbolic execution engine create symbols that have ranges for values when the program meets branches. It may seem complicated but the method is used by people intuitively to verify programs manually. In other words, according to the conditions of branches in the program, it researches to find out what values are assigned for the variables. The symbolic execution engine applies this principle to the inspection engine to see the path conditions in the entire program (Zhang *et al*., 2016).

Given the use of symbolic execution, it is possible to dramatically improve the code coverage of the black-box analysis. For example, if a particular path has security vulnerabilities, it continues to be vulnerable if one cannot inspect for such a path. Finding a vulnerability that appears only in very specific conditions is difficult, because if it is easy to know the specific path and the value which cause security vulnerability, it can also easily be exploited.

The symbolic execution is basically expressed as an unknown quantity instead of the concrete value. If the program meets the branching statement, it creates symbolic values for the routes on both sides. That is it is a way to figure out what values each variable has at each point after watching the conditions of several branching statements. The symbolic execution engine applies this principle to the program to see and follow the route condition that the engine accumulates. That is we can dramatically improve the code coverage if the dynamic analysis is done by the symbolic execution.

Let's assume the sample code is that in Algorithm 1. The sample code has 4 kinds of branch cases. It is not easy to create the data set (x, y) to execute all of the S0, S1, S3 and S4 which are all routes for the security vulnerability test.

**Algorithm 1; Sample code:**

```
Void test (int X, int y) {
  if (X>0) {
    if (y = = hash (X))
     S0
    else
     S1
  if (X>3 and and y>10)
   S3
  else
   S4
  }
}
int hash (X) {
  if (0< = X< = 10) retrurn X*10
  else return 0
}
```

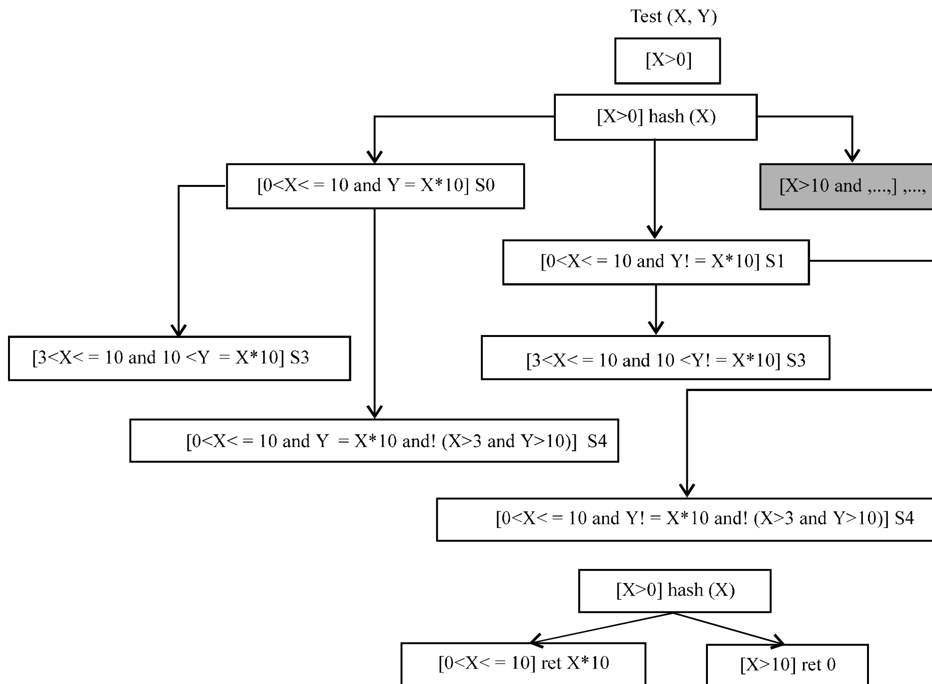


Fig. 1: Symbolic table

The symbolic execution does not assign the value to the variables but assigns the symbolic input during the program test. If we meet the branching statement while loading the program, the symbolic value is furcated by the constraint. If we fulfill the symbolic execution engine for the sample and source, the symbolic table (Fig. 1) is printed out. We can know what symbols (of X and Y) are assigned to each value and the constraint is set for each branch. By interpreting the constraint, we can find the test value for each branch.

If the input value is calculated for this output tree, the result is shown in Fig. 1. It is relatively easily to create a way to test the logic of each branch. That is, if the finally created data set is performed, the coverage of almost all test routes is provided.Test set obtained by the constraint solver:

- Test (1, 10) reaches S0
- Test (1, 0) reaches S1
- Test (4, 11) reaches S3
- Test (1, 10) reaches S4

The constraint table created by the symbolic execution engine is a result interpreted by the constraint solver as Eq. 3. If the route including the security vulnerability is discovered, we can hand over that data set to the smart fuzzing module to reenact it. If the symbolic execution engine and the constraint solver are combined, the maximum effect can be created by using the minimum test cases. This algorithm enables us to reduce the number of automated input tests and to find a security weak point that is not found easily. The symbolic engine does not require many test inputs and enables us to test by only a few input values (Kim *et al*., 2016).

## MATERIALS AND METHODS

### Automatic weak-code detection technique

**The architecture of a weak-code detection engine:** Figure 2 is a diagram of a weak-code detection engine for symbolic vulnerability analysis of open-source software. The weakly code detection engine for symbolic vulnerability analysis of open-source software consists of a source-file loading module, a compile library analysis module, a compiling module, a class-loading module, a byte-code analysis module, a method hierarchy tree generation module, an initial analytical target detection module, an analytical target method detection module, an analysis priority method list generation module and a DB for analysis and detection information.

The source-file loading module uploads the user's analytical target file and loads the analytical target stored in the source file repository on a project-by-project basis. The compile library analysis module applies libraries that

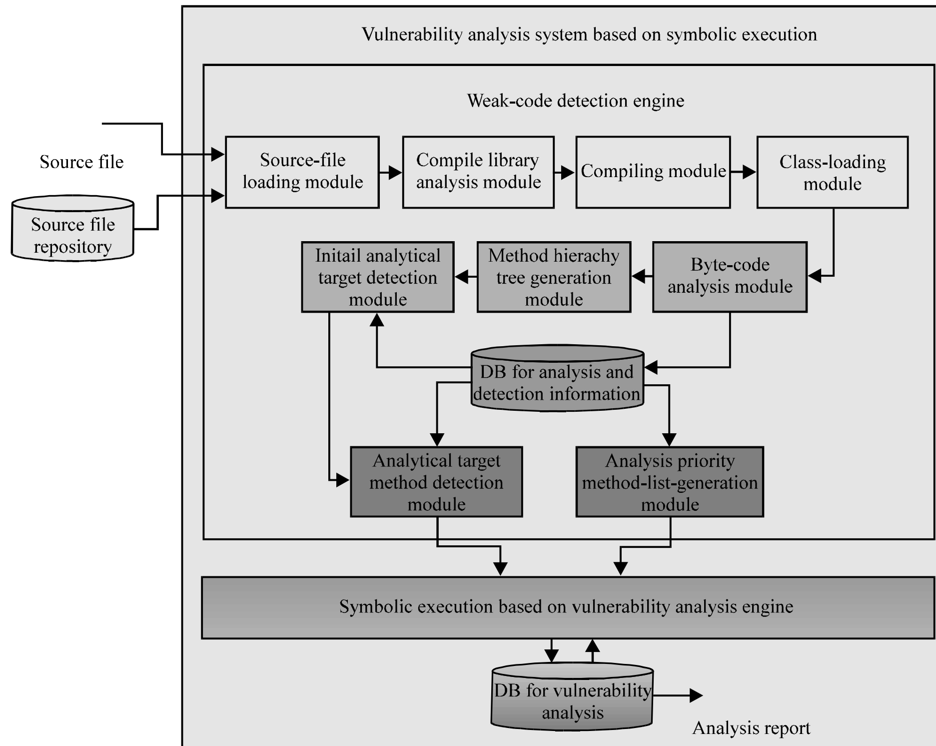


Fig. 2: A diagram of a weak-code detection engine for symbolic vulnerability analysis

are in accordance with the compiled source file by analyzing the types of compile libraries and the compiling module compiles the relevant source file. The class-loading module reads the class files created after compiling the source file. The byte-code analysis module reads the byte-codes of each class of the source file, collects and analyzes the necessary information from what the byte-code includes and stores the relevant information in the DB to create the structure and flow of the source file.

The method hierarchy tree generation module establishes the hierarchy tree, based on the DB information stored by means of byte-code analysis to understand the dependence or subordinate relationship between each class and method. The initial analytical target detection module detects methods that have potential risks to create a primary list of such methods. The analytical target method detection module detects the target methods by pattern matching the weak pattern list defined in the analysis and detection information DB and recognizes and categorizes the detected methods by the numbers and types of parameters.

The analysis priority method-list-generation module evaluates the complexity to enable prioritization, based on the information detected by the pattern-matching-based analytical target method detection module described above. The DB for analysis and detection information stores the previously mentioned byte-code analysis information and defines and stores the detection conditions of the initial analytical target detection module and the detection patterns of the analytical target method detection module.

**Weak-code detection process:** Figure 3 describes the functions performed by each module of the weak-code detection engine and the processing scheme. First, the target file of weak-code detection which is the target of the vulnerability analysis is loaded. The package information, including the directory information that allows more efficient management of interrelated classes, the import information including the library information that is used for analytical target file, the option information that is used for compiling the source path information including the location of the relevant source file and the library information included in the project is also loaded along with the analytical target file. Then by means of the compile library analysis, the type of compile library is identified and the corresponding compile library is applied where it make judgments on whether the library needed for source-file compiling is the compile library

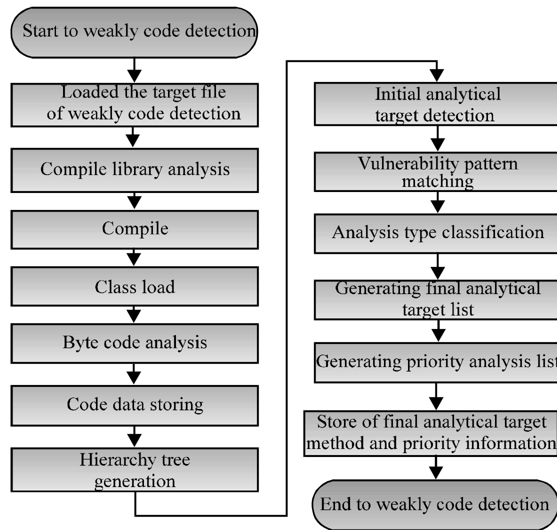


Fig. 3: A diagram of weak-code detection engine and the processing scheme

according to the direct importation of the developer or whether the compile library must be compiled with the plug-in type separated from a separate download of a library and applies the compile library that is correct for the source file.

The relevant source file is then compiled into the relevant compile library. Later, the classes created by the compiling are read by means of class load. By reading the byte-code of each loaded class file by means of byte-code analysis, the information on the calling class name, calling method name, calling location of the line, called class name, called method name, called location of the line, calling return type of method and the user class is analyzed and collected as information included in the class file.

The information collected by byte-code analysis is stored in the analysis and detection information DB by means of code data storing. When storing this, an analysis is to define the correlation of each source file and method and it is stored in the detection information DB to create the hierarchy tree and the logic inside the methods is also codified and stored. Then, the hierarchy tree is generated by referring to the information stored by the byte-code analysis by means of hierarchy tree generation and the tree data that allows knowing the program flow of the source file is generated according to the dependent or subordinate relationship between the classes or methods. When the hierarchy tree is generated, the first analytical target methods are detected by means of the initial analytical target detection. At this point, the three methods are detected and the method list is generated to detect the relevant method. The three methods are:

- The method of the class which is located at the end of the hierarchy tree structure
- The method that has a parameter in case the class does not have a method, although it is located at the end of the hierarchy tree structure
- The method that has more than four logics in between the start and the return

In the reason for setting four logics or more is that the experiment showed that the probability of finding vulnerabilities was greatest when there are four or more logics in between the start and the return. For the methods that are detected by means of the initial analytical target method detection, vulnerability pattern matching is performed and the corresponding methods are selected and extracted again. Using the weak pattern list predefined in the analysis and detection information DB, vulnerability pattern matching is performed to detect the secondary targets for method analysis. At this point, the weakly pattern list to be matched includes when there is a thread when there is a formula regarding the parameter and when the parameter is conditional. In other words, the methods that are in accordance with the following weak pattern list are detected through the pattern matching explained above:

- Methods that have threads
- Methods that have formula regarding the parameters
- Methods that have conditional parameters

An analysis type classification is performed to recognize and categorize the types of methods that are detected by means of vulnerability pattern matching in terms of the numbers and types of parameters. After the secondary analytical target methods are detected and the analytical types of the methods are categorized, a final method list is generated and stored in the analysis and detection information DB.

Then, a priority analysis list is generated based on the list of the final analytical target methods that is stored in the analysis and detection DB to determine the order of methods to be analyzed. The order of priority is defined in terms of the complexity of the method logic. The complexity is numerically defined based on the following criteria for the parameter type:

- If the factor from the parameter goes through the calculation formula
- How many branch statements are affected
- When the parameter goes through the formula
- What kind of arithmetic operations are used
- Whether or not the return value is determined by the factor of the received parameter and
- Whether there are threads inside the method affecting the external condition (return value)

The user can directly define the complexity value in the analysis priority method list generation module which allows the user to also determine what method on the priority list will undergo the vulnerability analysis. Lastly, the final analytical target method information and the priority information are saved in the analysis and detection information DB. The weak-code detection is then finished. The stored information is transferred to the symbolic execution security vulnerability inspection and the analysis engine to perform a vulnerability inspection of the target methods that the user predefined.

**Risk calculation of target methods:** Using the vulnerability pattern matching defined in subchapter B, risk calculation is performed for the methods that are the targets of the secondary analysis. The purpose of the risk calculation is to analyze the relationship between it and the actual vulnerability analysis results and to report to the open-source developers the riskiness of methods in the open-source that are currently coded:

- $M_{Tr}$; Method including threads
- $M_{P_cal}$; Method having formula regarding parameters
- $M_{P_con}$; Method having conditional parameters
- $N(M_k(1's)_{Analysis})$; Number of primary analytical target detection methods regarding source code k
- $W(M_{Tr})_k$; Riskiness of method having threads regarding source code k
- $W(M_{P_cal})_k$; Riskiness of method having formula regarding parameters of source code k
- $W(_{MP_con})_k$; Riskiness of method having conditional parameters regarding source code k

$$W\left(M_{Tr}\right)_k = \frac{\sum_{i=1}^{n} N\left(M_{Tr}(i)\right)}{N\left(M(1'S)\text{Analysis}\right)} \quad (1)$$

$$W\left(M_{P_{cal}}\right)_k = \frac{\sum_{i=1}^{n} N\left(M_{P_{Cal}}(i)\right)}{N\left(M(1'S)\text{Analysis}\right)} \quad (2)$$

$$W\left(M_{P_{con}}\right)_k = \frac{\sum_{i=1}^{n} N\left(M_{P_{Cal}}(i)\right)}{N\left(M(1'S)\text{Analysis}\right)} \quad (3)$$

As in the formulas above by means of the vulnerability pattern matching in terms of open-source k, the riskiness of the three methods which are the secondary analytical target is calculated and the open-source riskiness of the current version is stored. When the risky method of the open-source k of the current version is corrected according to the vulnerability analysis results, the new version of the open-source is re-investigated to calculate the riskiness of each and the results are stored. Therefore, as the vulnerability analysis and riskiness calculation is repeated, the riskiness of the three methods could be reduced by the developer's source code correction. The relationship between the calculated riskiness and the actual vulnerability analysis result will be discussed in subchapter 5.2, the experimental evaluation.

## RESULTS AND DISCUSSION

**Implementation:** A prototype symbolic execution-based vulnerability analysis tool was implemented to test the proposed technique. The system to carry this test out had a quad-core CPU of 2.66 GHz, 4 GB of RAM, 500 GB HDD and the Windows 7 operating system and the analysis tools were installed. A JSP page was created to report the results and interlocks of each analysis tool. Figure 4 shows the Java source code used for this test.

Figure 5 and 6 describe the log data that show the vulnerability analysis of the Java source code that was vulnerable because of the 'CWE-383 deadlock encountered, after the weak-code extraction using the proposed technique as a vulnerability analysis tool.

In Fig. 5, part a is the analysis option and b is the analytical target extracted by means of the weak-code which is the class and method of the analysis start. Part c is the calling of the analysis thread which shows the actual flow of data through the symbolic execution engine.

In Fig. 6, part d shows the vulnerability analysis which describes an error that occurred in the relevant thread. Part e is the content of the analysis which found an error at Thread Test.Java Line 27 after performing 'call stack,' an error received from the wait of ThreadTest.Java Line 7. Part f is the analysis result which shows the vulnerability in terms of CWE-383 deadlock encountered.'

**Test and evaluation:** By using the vulnerability pattern matching, we have conducted an actual vulnerability

**ThreadTest.java**

```java
public class ThreadTest {
    private volatile boolean active = false;
    private volatile int i = 0;

    public synchronized void mWait() throws InterruptedException {
        while (!active) {
            wait();
        }
        active = false;
        System.out.println("print\t" + i);
    }

    public synchronized void mNotify() {
        active = true;
        notifyAll();
        i++;
        System.out.println("after notify\t" + i);
    }

    private static class WaitCaller extends Thread{
        private final ThreadTest a;
        public WaitCaller(ThreadTest a) {
            this.a = a;
        }
        public void run() {
            try {
                a.mWait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    private static class NotifyCaller extends Thread{
        private final ThreadTest a;
        public NotifyCaller(ThreadTest a) {
            this.a = a;
        }
        public void run() {
            a.mNotify();
        }
    }
    public static void main(String[] args)  {
        final ThreadTest a = new ThreadTest();
        final int cnt = 2;
        for (int i = 0; i < cnt; i++) {
            new WaitCaller(a).start();
        }
        for (int i = 0; i < cnt; i++) {
            new NotifyCaller(a).start();
        }
    }
}
```

Fig. 4: Java source code used for the test

```
Console
Tomcat v7.0 Server at localhost (systest) [Apache Tomcat] C:₩Program Files₩Java₩jdk1.8.0_60₩bin₩javaw.exe (2016. 12. 8. 오후 4:51:10)
[DEBUG]=============================================================== error 1
[DEBUG]
[DEBUG]gov.nasa.jpf.vm.NotDeadlockedProperty                                    ⓓ
[DEBUG][CWE-383]deadlock encountered:

[DEBUG]  thread ThreadTest$WaitCaller:{id:2,name:Thread-2,status:WAITING,priority:5,isDaemon:false,lockCc

[DEBUG]
[DEBUG]=============================================================== snapshot #1
[DEBUG]                                                                          ⓔ
[DEBUG]thread ThreadTest$WaitCaller:{id:2,name:Thread-2,status:WAITING,priority:5,isDaemon:false,lockCour
  waiting on: ThreadTest@15b
[DEBUG]  call stack:
[DEBUG] at java.lang.Object.wait(Object.java)
[DEBUG]java.lang.Object.wait(Object.java)
[DEBUG] at ThreadTest.mWait(ThreadTest.java:7)
[DEBUG]ThreadTest.mWait(ThreadTest.java:7)
[DEBUG] at ThreadTest$WaitCaller.run(ThreadTest.java:27)
[DEBUG]ThreadTest$WaitCaller.run(ThreadTest.java:27)
[DEBUG]
[DEBUG]
[DEBUG]=============================================================== thread ops #1
[DEBUG]
    2      trans      insn          loc                   : stmt
------- ---------------------------------------------------------
W:15b      273      invokevirtual ThreadTest.java:7 : wait();
  S          2           runstart ThreadTest.java:0
[DEBUG]
[DEBUG]=============================================================== results      ⓕ
[DEBUG]
[DEBUG]error #1: gov.nasa.jpf.vm.NotDeadlockedProperty " [CWE-383]deadlock encountered:    thread ThreadT
[DEBUG]
[DEBUG]=============================================================== statistics
[DEBUG]
[DEBUG]SerialNumber:        SYM1481184579776
[DEBUG]elapsed time:        00:00:00
[DEBUG]states:              new=275,visited=310,backtracked=554,end=13
[DEBUG]search:              maxDepth=39,constraints=0
[DEBUG]choice generators:   thread=274 (signal=27,lock=42,sharedRef=90,threadApi=4,reschedule=111), data=€
[DEBUG]heap:                new=725,released=1591,maxLive=376,gcCycles=559
[DEBUG]instructions:        7503
[DEBUG]max memory:          737MB
[DEBUG]loaded code:         classes=64,methods=1430
[DEBUG]
[DEBUG]=============================================================== search finished: 16. 12. 8 오후5:09
```

Fig. 6: Symbolic execution vulnerability analysis operation 1

```
Console
Tomcat v7.0 Server at localhost (systest) [Apache Tomcat] C:₩Program Files₩Java₩jdk1.8.0_60₩bin₩javaw.exe (2016. 12. 8. 오후 4:51:10)
[DEBUG]analysis path :D:\eGovFrameDev-3.5.1-64bit\workspace\symtest\sysTest
[DEBUG]Running Symbolic Core ...
[DEBUG]symbolic.dp=choco
[DEBUG]symbolic.string_dp_timeout_ms=0
[DEBUG]symbolic.string_dp=none
[DEBUG]symbolic.choco_time_bound=30000
[DEBUG]symbolic.max_pc_length=2147483647                                  ⓐ
[DEBUG]symbolic.max_pc_msec=0
[DEBUG]symbolic.min_int=-100
[DEBUG]symbolic.max_int=100
[DEBUG]symbolic.min_double=-8.0
[DEBUG]symbolic.max_double=7.0
[DEBUG]Symbolic core system v8.0 (rev 31+) - (C) 2015 OpenSVS, INC. All Rights Reserved.
[DEBUG]
[DEBUG]
[DEBUG]=============================================================== system under test
[DEBUG]
[DEBUG]ThreadTest.main()                                                  ⓑ
[DEBUG]
[DEBUG]=============================================================== search started: 16. 12. 8 오후5:09
[DEBUG]
[DEBUG]after notify     1
[DEBUG]print    1
[DEBUG]after notify     2
[DEBUG]print    2
[DEBUG]print    2
[DEBUG]print    2
[DEBUG]print    2
[DEBUG]print    2
[DEBUG]after notify     2
[DEBUG]after notify     2
[DEBUG]after notify     2
[DEBUG]after notify     2                                                 ⓒ
[DEBUG]after notify     2
[DEBUG]after notify     2
[DEBUG]after notify     2
[DEBUG]print    2
[DEBUG]print    2
[DEBUG]after notify     2
[DEBUG]after notify     2
[DEBUG]after notify     2
[DEBUG]after notify     2
[DEBUG]after notify     2
[DEBUG]after notify     2
[DEBUG]after notify     2
```

Fig. 7: Symbolic execution vulnerability analysis operation 2

Table 1: Vulnerability analysis results of total, primary and secondary target method analyses

| | Vulnerability analysis result | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Total source-code analysis | | | | Primary target method analysis | | | | Secondary Target method analysis | | | |
| Group experiment no. | Riskiness | Vulnerability detection rate | Detection time (sec) | Vulnerability accuracy | Riskiness | Vulnerability detection rate | Detection time (sec) | Vulnerability accuracy | Riskiness | Vulnerability detection rate | Detection time (sec) | Vulnerability accuracy |
| Primary 200 files group analysis (avg.) | - | 0.32 | 3.98 | 0.39 | 0.64 | 0.52 | 1.25 | 0.42 | 0.48 | 0.52 | 1.01 | 0.41 |
| Secondary 200 files group analysis (avg.) | - | 0.51 | 4.75 | 0.45 | 0.45 | 0.51 | 2.21 | 0.55 | 0.55 | 0.41 | 2.02 | 0.55 |
| Tertiary 200 files group analysis (avg.) | - | 0.39 | 5.01 | 0.25 | 0.62 | 0.49 | 2.59 | 0.30 | 0.72 | 0.69 | 1.92 | 0.45 |

* Juliet code can be downloaded from http://samate.nist.gov/SARD/testsuite.php group analysis (avg.)

detection rate experiment by repeatedly calculating the riskiness of three methods that are the objects of the secondary analysis. The open-source file used for this experiment was Juliet code and a total of 600 Java files which consisted of the primary, secondary and tertiary groups of 200 files each. Table 1 shows the analysis results of the total source codes, primary source codes and secondary source codes among the Juliet codes used for this experiment. The result of the total source-code analysis did not include the calculation and application of riskiness but they were included in the primary and secondary analyses. The vulnerability accuracy shows whether or not the detected vulnerabilities are actually vulnerable codes in other words, if they were exploitative and these accuracy results include information on detection errors.

As shown in the result, the primary method analysis showed higher values than the total source code analysis and the secondary target method analysis result showed greater values than the primary target method analysis in terms of vulnerability rate and vulnerability accuracy. The time taken for the vulnerability detection was the longest in the total source code analysis, followed by the primary target method analysis and the secondary target method analysis.

## CONCLUSION

In this study, we suggested the technique of extracting the target and the area of weak-code when performing vulnerability analysis by means of symbolic execution analysis to inspect and analyze the vulnerabilities of open-source software. The suggested technique is as follows: a tree structure of the classes and methods within the source code of open-source software that is subject to security vulnerability analysis is created, then the DB of the target condition is established by identifying the data types and configuration patterns of the methods and it is examined by means of the system. If the condition is met, the corresponding method that is subject to vulnerability analysis is automatically extracted. The proposed technique is to extract and analyze the methods with high risk, instead of making a total inspection of source files when analyzing the security vulnerabilities by means of symbolic execution which reduces analysis time and system load by extracting and inspecting only the weak-codes by means of the analysis of byte-codes of the source file, hierarchy tree generation, extraction of initial analytical targets, pattern matching-based target method extraction and analysis priority method list generation to extract the methods having high risk. Also, by using the defined vulnerability pattern matching, the riskiness of the secondary analytical target methods is calculated and applied to report to the open-source developer the degree of riskiness of the corresponding coded methods of the open-source version and it can provide a standard for the vulnerability analyzers to selectively analyze the vulnerability in accordance with the vulnerability riskiness for an effective vulnerability analysis.

The prototype analytical tool of the proposed technique to inspect the vulnerabilities of open-source software was implemented and we proved the excellence of the performance of the suggested technique by means of the experimental evaluation.

## REFERENCES

Allodi, L. and F. Massacci, 2014. Comparing vulnerability severity and exploits using case-control studies. ACM. Trans. Inf. Syst. Secur., 17: 1-20.

Chen, T., X.S. Zhang, C. Zhu, X.L. Ji and S.Z. Guo *et al*., 2013. Design and implementation of a dynamic symbolic execution tool for windows executables. J. Software Evol. Process, 25: 1249-1272.

Ernst, M.D., 2003. Static and dynamic analysis: Synergy and duality. Proceedings of the International Conference on Software Engineering (ICSE) and Workshop on Dynamic Analysis (WODA'03), May 03-10, 2003, University of Portland, Portland, Oregon, pp: 24-27.

Kebbal, D., 2006. Automatic flow analysis using symbolic execution and path enumeration. Proceedings of the 2006 International Workshops on Parallel Processing, August 14-18, 2006, IEEE, Columbus, Ohio, pp: 1-8.

Khalili, A., A. Sami, M. Azimi, S. Moshtari and Z. Salehi *et al*., 2016. Employing secure coding practices into industrial applications: A case study. Empirical Software Eng., 21: 4-16.

Kim, J.H., M.C. Ma and J.P. Park, 2016. An analysis on secure coding using symbolic execution engine. J. Comput. Virol. Hacking Tech., 12: 177-184.

Microsoft Corporation, 2016. Benefits of the SDL. Microsoft Corporation, Redmond, Washington, USA. https://www.microsoft.com/en-us/sdl/about/benefits.aspx.

Ransbotham, S. and S. Mitra, 2013. The Impact of Immediate Disclosure on Attack Diffusion and Volume. In: Economics of Information Security and Privacy, Schneier, B. (Ed.). Springer, New York, USA., ISBN:978-1-4614-1980-8, pp: 1-12.

Zerkane, S., D. Espes, P.L. Parc and F. Cuppens, 2016. Vulnerability Analysis of Software Defined Networking. In: Foundations and Practice of Security, Cuppens, F., L. Wang, N. Cuppens-Boulahia, N. Tawbi and J. Garcia-Alfaro (Eds.). Springer, Switerzland, ISBN:978-3-319-51965-4, pp: 97-116.

Zhang, B., C. Feng, B. Wu and C. Tang, 2016. Detecting integer overflow in Windows binary executables based on symbolic execution. Proceedings of the 17th IEEE-ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel-Distributed Computing (SNPD'16), May 30-June 1, 2016, IEEE, Shanghai, China, ISBN:978-1-5090-0804-9, pp: 385-390.

Zhejun, F., Y. Zhang, Y. Kong and Q. Liu, 2013. Static detection of logic vulnerabilities in Java web applications. J. Secur. Commun. Netw., 7: 519-531.